Data structures based on k-mers for querying large collections of sequencing datasets

Camille Marchet, Christina Boucher, Simon J Puglisi, Paul Medvedev, Mikaël Salson, Rayan Chikhi

Data structures based on k-mers for querying large collections of sequencing datasets

Camille Marchet[1], Christina Boucher[2], Simon J Puglisi[3], Paul Medvedev[4,5,6], Mikaël Salson[1], and Rayan Chikhi[7]

[1]Université de Lille, CNRS, CRIStAL UMR 9189, Lille, France
[2]Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA
[3]Department of Computer Science, Helsinki Institute for Information Technology HIIT, FI-00014 University of Helsinki, Helsinki, Finland
[4]Department of Computer Science, The Pennsylvania State University, University Park, USA
[5]Department of Biochemistry and Molecular Biology, The Pennsylvania State University, University Park, USA
[6]Center for Computational Biology and Bioinformatics, The Pennsylvania State University, University Park, USA
[7]Institut Pasteur & CNRS, C3BI USR 3756, Paris, France

Correspondence: ***camille.marchet@univ-lille.fr***

Section 1: Introduction

Over the past decade the cost of sequencing has decreased dramatically, making the generation of sequence data more accessible. This has led to increasingly ambitious sequencing projects. For example, the completion of the 1,000 Genomes project (1) led to the advent of the 100,000 Genomes (2), which has since been completed. There are dozens of other large-scale sequencing projects underway, including Geuvadis (3), GenomeTrakr (4), and MetaSub (5). There is now an overwhelming amount of public data available at EBI's European Nucleotide Archive (ENA) (6) and NCBI's Sequence Read Archive (SRA) (7). The possibility of analyzing these collections of datasets, alone or in combination, creates vast opportunities for scientific discovery, exceeding the capabilities of traditional laboratory experiments. For this reason, there has been a substantial amount of work in developing methods to store and compress collections of high-throughput sequencing datasets in a manner that supports various queries. These methods are now fundamental to several bioinformatic applications, as we detail in Section 2.

In this paper, we use the term dataset to refer to a set of reads resulting from sequencing an individual sample (e.g., DNA-seq, or RNA-seq, or metagenome sequencing). Sequencing is routinely performed not only on a single sample but on a collection of samples, resulting in a collection of datasets. For instance, 100,000 human genomes were sequenced for Human Genome Project and over 300,000 bacteria strains were sequenced for GenomeTrakr. One basic query that is fundamental to many different types of analyses of such collections of datasets can be formulated as follows: given a sequence, identify all datasets in which this sequence is found. For example, consider the problem of finding a RNA transcript within a collection of RNA-seq datasets. Similarly, we can ask to find which datasets contain a specific DNA sequence, such as a gene or a non-coding element, in a collection of bacterial strain genomes.

Given the size of many collections and datasets, several different paradigms for storing them in a manner that they can be efficiently queried have been developed – many of which continue to be extended and explored. One paradigm is to store and index datasets as sets of k-length substrings, which are referred to as k-mers. We will refer to collections of datasets as sets of k-mer sets. The methods that use this paradigm first build an index of all k-mer sets, and support the basic query described above by splitting the query sequence into k-mers and determining their presence or absence in the index.

As we will discuss in this survey, this paradigm proves to be useful in several ways. First, sets of k-mers is a more concise representation of the set of sequences of the samples, as they abstract some of the redundancy inherent to the high sequencing coverage. Second, genetic variation and sequencing errors can be dealt with in a more-efficient, albeit less accurate, way than using sequence alignment. Inexact pattern matching can be simply realized by examining the fraction of matching k-mers within the query sequence. However, representing and storing data using a k-mer index comes with some loss of information since it only gives information for each constituent k-mer of a sequence – rather than information about the entire sequence. Hence, in most cases a k-mer index does not provide exact answers for queries of longer sequences (e.g., whole transcript or entire gene) but instead provides an approximation.

One can view k-mer-based indexes as building blocks for the topic of the present survey: the storage of k-mers coming from multiple datasets, i.e., sets of k-mer sets. A key aspect of storing sets of k-mer sets comes from the observation that sequencing experiments that are collectively analyzed typically share a large fraction of k-mers. Therefore, significant space savings can be achieved by the identification and clever storage of this redundant information. Here, we focus on the methods underlying the different building blocks (Figure 1) of sets of k-mer sets structures. We review the different properties, the types of queries, and the computational performance which they offer. We highlight similarities of methods based on commonalities between building blocks where it is appropriate.

Section 2: Biological applications

Several biological and biomedical questions can be addressed by querying sequence databases. Transcriptomics was one of the first specific applications to be described. Solomon

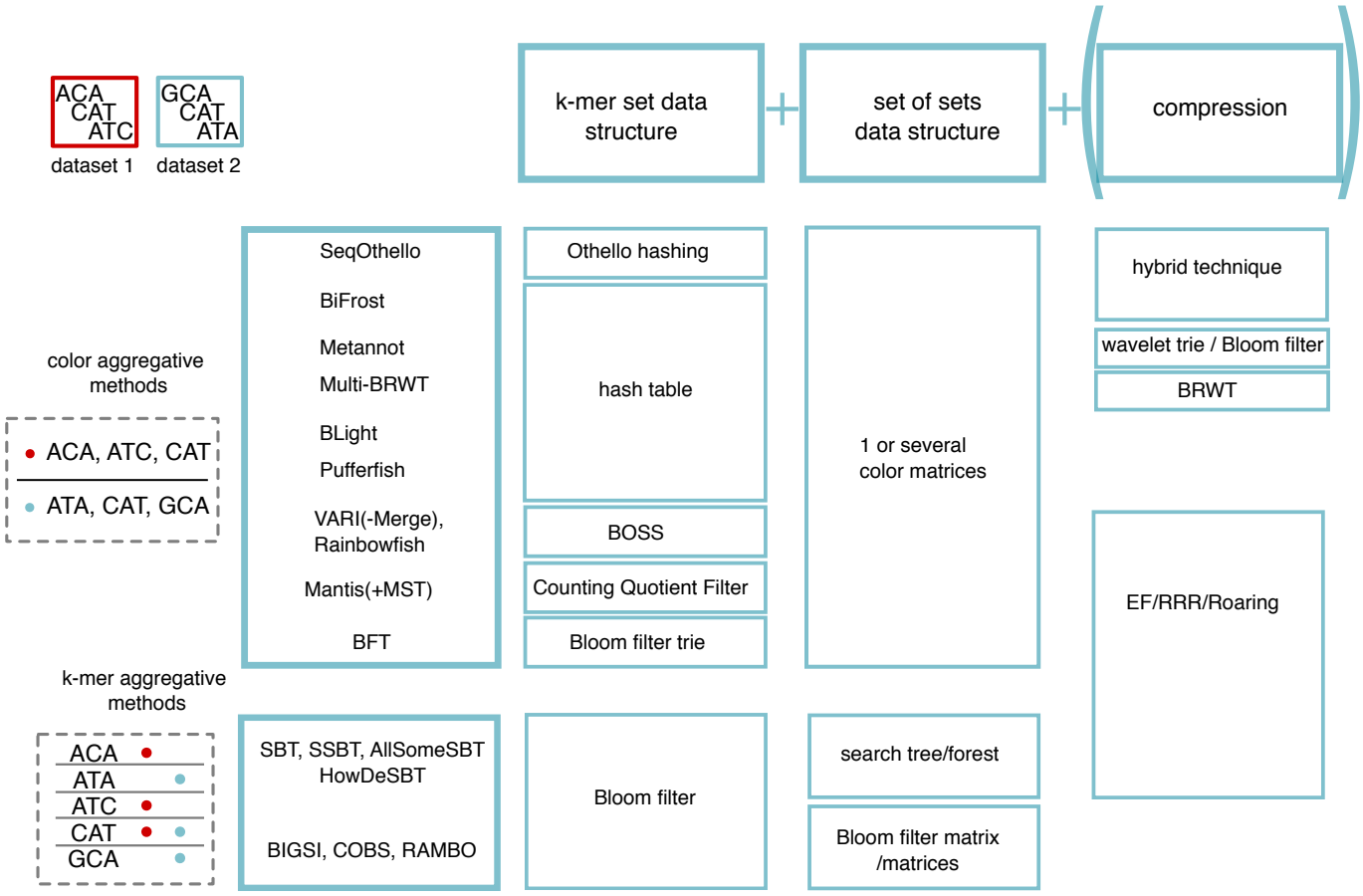


Fig. 1. Overview of set of k-mer sets building blocks. We classified strategies in color-aggregative approaches, and k-mer aggregative approaches. The top row of the figure indicates the general categories of components: a k-mer set, possibly combined with a de Bruijn graph representation; a way to combine multiple sets together; optional compression. Each next row describes a general strategy adopted by one or more methods.

and Kingsford gathered more than 2,000 samples of human RNA-seq, consisting of blood, brain, and breast tissue samples from SRA (8). Since its introduction, this database kept growing, roughly doubling each year. This leads to the possibility of identifying conditions which express isoforms by associating transcripts to tissues. Similarly to tissue-specific associations, one can envision the numerous benefits of comparing patient cohorts in order to understand differences in pathologies or impact of medication. For instance, using RNA-seq for functional alterations profiling has become more frequent in cancer research (9). Thus, vast programs such as The Cancer Genome Atlas (TCGA) (10), provide RNA-seq samples from a variety of cancer types. Yu et al. (11) showed how to investigate gene-fusion using a set of k-mer sets by first creating an index of all tumor samples from the TCGA. Then they considered documented fusion events and their corresponding k-mer signatures, and screened the index to detect these signatures. They confirmed some fusions and reported some novel ones. Fusion transcripts provide interesting targets for cancer immunotherapies since they are susceptible to exhibit tumor-specific markers.

Several papers demonstrate how sets of k-mer sets could help mine and analyze collections of microbial samples or genomes, whether they be strains of the same genera (e.g., 16,000 strain of *Salmonella*), microbiomes (e.g., 286,997 genomes from the human microbiome), or more extensive microbial data (e.g., 469,654 bacterial, viral and parasitic datasets from the ENA). For example, GenomeTrakr (4) was developed to coordinate international efforts in sequencing whole genomes of food-borne pathogens. Indexing and querying this and other databases could lead to improved surveillance of pathogenic bacteria, and thus, elucidate the effectiveness of interventions that attempt to control them. Subsequently, k-mer indexes have been used to follow the spread of antimicrobial resistance (AMR) genes and plasmids across bacterial populations. Bradley et al. (12) also searched for plasmid sequences bearing AMR and initiated a study in an index containing a variety of microbial genomes. They identified some of these plasmids spread across different genera. Lastly, an effort was proposed to build a comprehensive human gut microbiome resource with the help of a set of k-mer set structure. Cultured genomes and metagenomics assembled-genomes from metagenomics samples were combined to create the Unified Human Gastrointestinal Genome index (13). This resource aims at being explored and enables looking for contigs sequences, genes, or genetic variants.

Beyond these applications, other topics start to be explored: integrated variant calling across large-scale gene, plasmid and transposon search (12, 14, 15), bacterial pan-genome in-

dexation (16), gene fusion and pan-cancer analysis (11).

Section 3: Building blocks

We view the storage of a set of k-mer sets as having various (optional) components, which we refer to as building blocks. See Figure 1 for an illustration of this view. In light of this, we cover the basic terminology and concepts that will be used throughout this survey in this section.

3.1 *k*-mer set data structures. As previously described, the main building blocks used in sets of k-mer sets representations are, unsurprisingly, schemes to represent k-mer sets. The methods surveyed in this work represent k-mers from sequencing datasets as sets, i.e. k-mers are de-duplicated and unordered. It will be enough to consider the data structures presented in this section as black-boxes that support all or a subset of the following operations:

- membership, i.e., testing whether an element is in the set;
- insertion and/or deletion of an element.

Set representations that support insertion and/or deletion are sometimes said to be dynamic, as opposed to static. However, methods to represent k-mer sets are not all equivalent in terms of features and performance. We briefly review their main characteristics in the rest of this section but refer the reader to the recent survey by Chikhi et al. (17) for further details.
Most methods rely on bit vectors to store the presence or absence of k-mers in datasets. A bit vector is an array of bits, e.g., 00101 represents a bit vector of length 5. A 0 is used to indicate that the k-mer is absent, and a 1 indicates that it is present. A bit vector could be used to record the datasets in which a given k-mer appears, or, alternatively, all the k-mers that are contained in a given dataset. However, with a growing number of datasets and k-mers, using plain binary vectors is generally too simplistic, and often compression or other tricks are also incorporated. One example is the Bloom filter (18), which is a way to store a set as a bit vector using many fewer bits than the naive approach.
Some methods view a k-mer set as a de Bruijn graph (DBG). In a DBG, vertices are k-mers and there exists a directed edge from vertex u to v if the the last $k-1$ characters of u are the same as the first $k-1$ characters of v. See Supplementary Box 2 for an example. In practice, these graphs are bi-directed in order to capture the double-stranded nature of DNA, but viewing them as directed usually simplifies the description of many methods without sacrificing any important aspects. These two views, k-mer sets and DBGs, are (in some sense) equivalent as they intrinsically represent the same information. Yet, in some applications it is more convenient to consider a graph representation, e.g. for genome assembly (19) and variant detection (20).
Data structures for representing k-mer sets (DBG or not) can further be divided into two categories: membership data structures and associative data structures. The first category only informs about the presence or absence of k-mers, e.g., as in the case of a Bloom filters. The second category associates pieces of information to k-mers, akin to how dictionaries link words to their definitions. Some examples of such data structures include hash tables and counting quotient filters (CQF) (21–23). See Supplementary Box 1 for an illustration.
Some data structures (e.g. CQF) can represent sets in an exact way; whereas others (e.g. Bloom filters) represent them in a probabilistic way, in that the structures can return false positives (i.e., meaning that a k-mer is sometimes reported as present in the set when in fact it is not present). These false positives lead to an over-estimation in the number of k-mers detected as present in a set. While this is an undesirable effect, it can be partly mitigated – as we will see in Section 4.2.1.
Finally, other methods for representing k-mer sets rely on full-text indices (e.g. BOSS, FM-index (24, 25)). Such indices store more information than just a set of k-mers, and support queries of sequences up to the read length over the sequences.
The main reason such advanced data structures are considered, instead of those provided in the standard libraries of programming languages, is space efficiency. Bloom filters and CQFs approximately require a byte for each element in the set, i.e., less than the size of the element itself. Similarly, succinct representations of DBGs (26–28), which are also representations of k-mer sets, aim for near-optimal space efficiency. The difference between the two approaches is the trade-off that they offer between space and accuracy. This is a crucial aspect as the volume of data typically exceeds what can be processed using unoptimized data structures.

3.2 Compression. To further optimize space usage, different compression techniques have been applied to sets of k-mer sets. Bloom filters and bit vectors are typically objects which are amenable to a number of compression techniques because they are represented in bits. Such objects can be sparse (i.e. when most of the bits are zero, such as when many k-mers belong to only a few datasets from the whole collection) and/or dense (i.e. when most of the bits are 1, such as when many k-mers are present in the same set of datasets). Compression methods exploit these properties. Different compression strategies are shown in Box 2.

Bit vector compression. Bit vectors can be efficiently stored using bit-encoding techniques that exploit their sparseness or redundancy. The most prevalent of the methods in this survey are RRR (29) or Elias-Fano (EF) (30–32) (see Box 2 (1)). The principle behind these is to find run of 0's and to encode them in a more efficient manner, reducing the size of the original vectors. Other techniques such as Roaring bitmaps (33) adjust different strategies to sub-parts of the vectors. Wavelet trees (34) generalized compression of vectors on larger alphabets (i.e., not just 0's and 1's but e.g. a's, b's, c's, etc). More advanced techniques deriving from the same concept were also proposed specifically for sets of k-mer sets (35).

Delta-based encoding. When two sets share many elements, it may be more advantageous to store only the differ-

ences between those datasets. For instance, rather than storing two (possibly compressed) bit vectors, one can only store the first bit vector explicitly along with a list of positions of differences for the second. This list of positions can itself be encoded as a bit vector, with a bit set to one if and only if the bit is at the location of a difference between the two vectors. This bit vector is expected to be more sparse than the original encoding, allowing better compression. Such a scheme is usually called "delta-based encoding" in the literature. An example is presented in Box 2 (2).

Hybrid techniques. In hybrid approaches, bit vectors are first separated into different buckets, where each bucket will contain bit vectors with similar features. Compression within each bucket is performed using a suitable technique selected from a pool of feature-specific ones (see Box 2 (3)).

Section 4: Aggregation techniques

As we illustrate in Figure 1 , some methods first index each dataset separately, and then build structures that allow retrieving the name of datasets in which a k-mer is present; whereas others start by representing the set of all k-mers from all datasets, and then attribute them to their datasets of origin (colors). We refer to the former as k-mer aggregative methods and the latter as color-aggregative methods. We will discuss both aggregation methods. We present the few methods that escape this categorization separately.

4.1 Color-aggregative methods. These methods start by indexing all k-mers across all datasets, and then group together (aggregate) datasets into what will be referred to as color sets. A practical advantage of color-aggregative methods is that a k-mer that appears in many samples will appear only once in the k-mer set. This greatly reduces redundancy in the representation. In this subsection, we give a brief background and history of the methods that fall into this category.

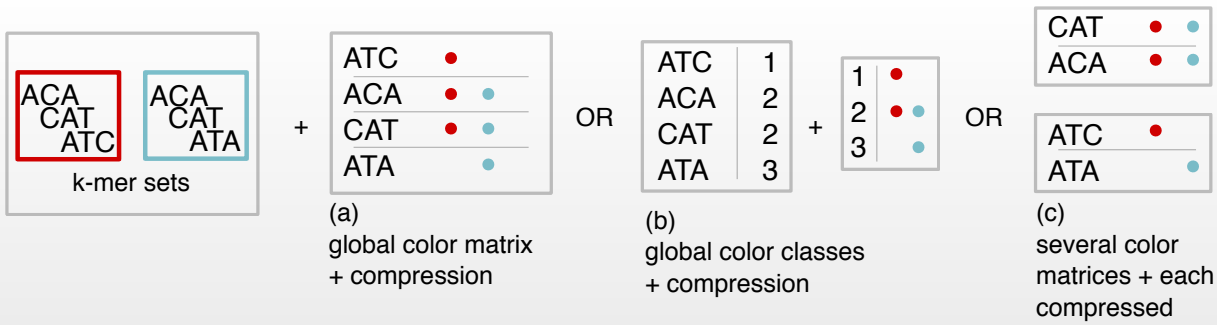
4.1.1 Representation of colors. A color of a given k-mer is frequently used in the literature to identify a dataset containing the k-mer, assuming each dataset is given a unique color. A k-mer may often be contained in multiple datasets and therefore, it is convenient to store the colors of a k-mer using a bit vector. Here, given order of the datasets, a 1 at position i in the bit vector indicates the presence of the k-mer in the i-th dataset, and 0 its absence. A color set is the set of colors associated to a k-mer. Given n k-mers and c datasets, the color matrix is a $n \times c$ matrix of bits which describe the presence or absence of each k-mer (in the rows) in each dataset (in the columns). For an example, see Box 1.

4.1.2 Background and method intuition. Iqbal *et al.* (20) defined the colored de Bruijn graph (cDBG) – which, moreover, is also the first color-aggregative method. Given a set of input datasets, each dataset is assigned a unique color, so that there are as many colors as datasets. When considering the union of all k-mers across all datasets, it is straightforward to think of using an associative structure to map k-mers to colors. cDBG are DBGs of the set of all pooled k-mers from all the samples (union graph), with each vertex labelled with the k-mer's color set. Colored DBGs allow traversing operations to find paths that contain undirected cycles (bubbles), and finding the individuals in the population that contain a shared or divergent path in each bubble.
Cortex implemented those properties, and its bubble-finding algorithm was one of the primary use cases of the cDBG, to detect biological genetic variation in a population without the use of a reference genome. Unfortunately, Cortex consumes an inordinate amount of RAM when the total number of distinct k-mers exceeds tens of billions. This main drawback motivated more recent works taking benefit from the cDBG.
We note that later in this survey, we will not restrain the term cDBG to the original work from Iqbal et al., but to any explicit DBG implementation that associates color sets to k-mers. Colored DBGs are only implemented in the class of color-aggregative methods. Though, as we will see later, some color-aggregative methods do not implement a cDBG.

4.1.3 Space-efficient color-aggregative methods. The first methods that improved upon the memory- and time- efficiency of Cortex were Bloom filter trie (BFT) (36) and Vari (24). These methods achieved a significant reduction in representation size via different strategies.
After the introduction of these methods, subsequent improvements were made with the development of Rainbowfish (37), Multi BRWT (35), Mantis (38), SeqOthello (11), Mantis+MST (23), and Vari-Merge (16). Most of these recent techniques rely on a more careful encoding of the colors of each k-mer, which commonly takes advantage of redundancy in the data. In many applications, such as human RNA-seq indexing (8), it is expected that many datasets share a large number of k-mers. This redundancy can be exploited to reduce the color encoding size, notably through color classes. When colors are seen as bit vectors, the color classes are simply de-duplicated bit vectors. Thus two k-mers having the same color set are associated to a single color class instead of two identical bit vectors (see Box 1). Compression may be achieved by representing the color matrix as a compressed bit-vector (see Box 2 (1)). The growing number of colors (or classes) motivated works to further reduce space through lossy compression, notably Metannot (39). The overall coloring strategies are also presented in Supplementary Box 5. We now describe each of these methods in more detail. With the exception of SeqOthello, all methods consider the underlying k-mer set representation to be a DBG.

◇ **Vari and Vari-Merge.** The succinct representation of the DBG used by Vari is referred to as BOSS (27), which is related to the Burrows Wheeler Transform (BWT). While we will not go into the technical details here (see Appendix and Supplementary Box 4 for more information), it is sufficient to see BOSS as a rearrangement of the original data that enables indexing and compression. In order to add color information in Vari, a compressed binary matrix stores which edges e_i of the DBG are present in which sample s_j. The matrix is constructed row-by-row, using the RRR bit vector representation for compression.

Box 1. Color coding



A **color matrix** represents the presence of n k-mers across c datasets (here $n = 5, c = 3$). Different schemes have been introduced to encode such matrices. In particular, a **color class** is a set of colors (or equivalently, a bit vector, or a line in the color matrix) that is common to one or more k-mers. Indeed, in the color matrix some k-mers (lines) may correspond to the same color class. One may "de-duplicate" the n lines of the color matrix into only $m < n$ color classes (here, $m = 4$). Then, color class identifiers are introduced as intermediaries between k-mers and color classes. (To go further, frequently used color classes can be represented using fewer bits by using small integers as identifiers).

Color-aggregative methods are generally composed of three parts. First, a set of all k-mers, built using either a succinct DBG or an ad-hoc representation. Second, (a) a correspondence between k-mers and colors in the form of a color matrix, color classes (b), or several color matrices (c). Finally, k-mer sets and/or colors may be further compressed (see Box 2).

Later, Vari-Merge (16) was introduced to construct colored DBGs for very large datasets. It first divides the data into smaller subsets, then constructs a succinct DBG (using Vari) for each, and lastly, succinctly merges then until a single one remains. Because of the merge procedure, it allows for the addition of new data.

◇ **Bloom Filter Trie.** BFT uses a different approach to storing the DBG than Vari but also aims at representing a DBG ((4) in Figure 1). BFT is built upon a trie: a particular tree structure allowing to store a set of strings along with associated data. Rather than using BOSS to construct and store the DBG, a trie is used to store the nodes and to associate labels. Bloom filters are also added in the trie for efficiency. BFT introduces the idea of color classes, i.e. non redundant sets of samples in which a k-mer is present (see Box 1 (b)). A label is assigned to each color class, and the color matrix becomes a table that associates each k-mer to its corresponding label.

◇ **Rainbowfish.** Rainbowfish mixes ideas from Vari and BFT. It uses a similar structure to Vari to store the k-mer set structure but stores color classes in a similar manner as in BFT. In addition, to obtain sparser bit vectors, it encodes the most frequently used classes using more zeros. As for compression, the color class and the label are represented as bit vectors that can be compressed.

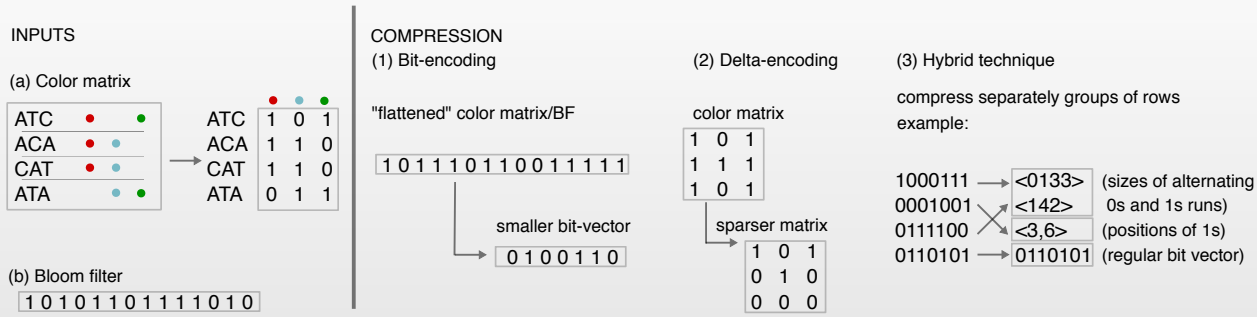
◇ **Mantis and Mantis+MST.** Mantis (38) introduces another strategy for storing the DBG in a space-efficient manner. (see strategy (3) in Figure 1). Mantis starts by building, for each dataset, an associative data structure on k-mers (CQF, see Supplementary Box 1). Initially, CQFs were introduced to record counts associated to k-mers but in Mantis, the structure instead stores color sets. Mantis then merges all CQFs in order to have the global information in a global filter.
An extension of Mantis takes advantage of the insight that many color classes are frequently similar to each other, since many k-mers occur in relatively similar sets of sequences. Thus Mantis+MST proposes to encode colors using only the differences between classes (delta-based encoding, see Box 2 (2)).

◇ **SeqOthello.** SeqOthello does not explicitly represent a DBG but rather stores a probabilistic set of k-mers using Othello hashing (see See strategy (6) in Figure 1). Like in any color-aggregative method, k-mers are associated with colors using regular bit-vectors. As seen in Subsection 4.1.1, similar presence or absence bit vectors are grouped in order to apply hybrid compression (see Box 2 (3) and Box 1 (c)). SeqOthello proposes to group similar color profiles, then use a suitable compression technique depending on the bit-sparsity of each group. Othello hashing can yield false positives when alien k-mers are queried (i.e., k-mers that are not present in the indexed collection). In other words, Othello hashing may wrongly associate a dataset to an alien k-mer, instead of correctly returning that such k-mer belongs to no dataset. For a query that consists of many k-mers (such as genes or transcripts) errors can be mitigated because false positives are unlikely to all point to the same dataset(s). This property will also be used in k-mer aggregative structures, and will be further discussed in a dedicated section.

◇ **Bifrost, Pufferfish, and BLight** Recently, another construction strategy for color-aggregation was proposed in Bifrost (40), which relies on two techniques: 1) in-memory construction of the DBG, 2) dynamic updates of the graph

Box 2. Compression of bit arrays

Colors matrices (as shown in (a), along with an equivalent bit vector representation) and Bloom filters (b) can be compressed. We present here some of the known techniques.



Bit encoding techniques (1): If working with a color matrix, rows of the color matrix are first concatenated. The resulting bit vector is then compressed losslessly into a shorter bit vector.
Delta-based encoding (2): Differences between rows in the matrix are encoded. One can e.g. encode the (column-wise) differences between and current row and the first row as 1's, and similarities with 0's. This results in a sparser matrix that can be further compressed using e.g. (1).
Hybrid compression technique (3) Nearly identical rows are grouped, and a representative is chosen for each group. Then, depending on whether bit-vectors associated to the color classes are sparse (little amount of 1s) or dense (high amount of 1s), different compression schemes are used.

and colors. It proposes an idea that is related to Mantis+MST: k-mers that are close in the DBG are more likely to share the same colors. Thus, several color matrices are built for distinct sequences of the DBG, instead of one for the whole graph. In each color matrix, the compression rate is expected to gain from the similarity of each color vector to one another.
Pufferfish (41) and BLight (42) rely on similar concepts to Bifrost. In particular, all three use a compacted DBG (introduced in e.g. (43, 44), see Supplementary Box 2), to more efficiently represent the sequences than a set of individual k-mers. In addition, we note these methods are similar to Mantis and SeqOthello in terms of the underlying hash-based strategies, and the differences are detailed in Supplementary Box 6.

◇ **Other methods.** Other methods have focused on specific aspects of color-aggregative methods memory optimization. Metannot (39) and Multi-BRWT (35) both construct the color matrix in a manner that is both compressed and dynamic but use a simple representation of the union graph where each edge (k-mer) is stored in a hash table. This graph representation allows it to be updated along with the color information – thus, the main contribution is not the data structure used to store the graph, but the one used to store colors. Metannot explores two strategies for color compression. First, they propose to use a wavelet trie. It allows to index the color matrix rows in nearly optimal compressed space. This strategy is combined with RRR on the rows ((1) in Box 2). Second, they use a lossy color representation with Bloom filters. To reduce false positives, color sets queries are corrected by taking the intersection with other color sets from neighboring k-mers in the DBG. Multi-BRWT improves upon standard bit-encoding representation (such as RRR, EF) by allowing for compression of both dimensions of the matrix simultaneously instead of compressing each row (or column) once at a time.

4.1.4 *Queries.*

Given the current implementations work with rather small k-mer sizes ($\sim$ 20 to 63), the query time bottleneck comes from the memory random accesses. Hash-based methods perform very fast queries: retrieving a k-mer requires only a constant number of accesses. The methods whose underlying DBG is BOSS (e.g., Vari, Rainbowfish, Vari-Merge) are expected to show a lower throughput. Indeed, the retrieval of a k-mer requires roughly k memory accesses.

4.1.5 *Additional properties.*

We observe that the features offered by any given data structure are strongly dependent on the choice of the data-structure for the underlying k-mer set.

◇ **Insertion or deletion of data.** Only a few methods focused on insertion and/or deletion of data. See Table S3. Frequently, space-efficient methods require that the whole index needs to be rebuilt when new data needs to be inserted into the index. This applies to SeqOthello, Pufferfish, or BLight because their hashing technique is static. Although it is conceptually possible that new data can be added to Vari, Rainbowfish, and Mantis, this feature is currently not implemented. One of the main advantages of Vari-Merge and Bifrost is that they allow new data to be added via merging. However, while Vari-Merge merges the entire index, Bifrost only dynamically adds sequences to the DBG, while colors need to be rebuilt.

As previously mentioned, Metannot and Multi-BRWT allow for both the addition and deletion of new data but do not compress the union graph.

◇ **Colored DBG implementation and traversal.** Vari, VariMerge, BFT, Bifrost and Rainbowfish explicitely implement cDBGs. They consider graph navigation, as initially proposed in Cortex. Features include retrieving paths (i.e., sequences of edges that connect distinct nodes), and bubble calling. Yet, we note that this aspect should be technically possible in all cDBG tools.

4.2 k-mer aggregative methods. We now turn to a completely different class of data structures. Unlike previously mentioned methods, k-mer aggregative methods do not pool k-mers from all datasets in order to build an index. Rather, they first process datasets separately, and then aggregate them in different ways to speed-up queries.

4.2.1 Background and method intuition. The design of these methods is motivated by the following use case. Let Q be a query set of k-mers to be searched in a collection of datasets $\mathcal{D}$, where each element of $\mathcal{D}$ represents a dataset. For example, Q may be the set of k-mers contained in a single transcript. The collection $\mathcal{D}$ can be seen as a set of k-mer sets. One seeks to find the sets $D_i \in \mathcal{D}$ such that $|D_i \cap Q|/|Q| \geq \theta$, with θ being a given ratio of k-mers, seen as a stringency parameter for the search. The value of θ can be set to allow for sequencing errors or mutations, e.g. between 0.7 and 0.9 (8). Whereas in the methods of Section 4, a query is a single k-mer whose color class we would like to know, in this context, the query is a set of k-mers, and we would like to know the datasets which contain at least a θ fraction of them. We will refer to the query of a single k-mer as a k-mer query and the query of a set of k-mers as a set query.
The methods in this section take explicit advantage of the motivating use case by storing the k-mers of each dataset in a Bloom filter (BF), i.e. one BF per dataset. Recall from Section 3 that a BF is a probabilistic data structure that sometimes return false positives; i.e. the BF may report that a k-mer belongs to a certain dataset when it really does not. However, in the setting of the set query, the θ parameter allows us to naturally absorb any potentially detrimental effect of such false positives. For the values of θ used in practice, the false positive rate of a BF for k-mer queries can be as high as 50% without degradation of the set query performance (8). Indeed, the false positive rate of a set query decreases exponentially with the number of k-mers (8, 45).
Thus, contrary to color-aggregative methods, these methods are pre-sized before the addition of k-mers. The size is chosen according to the desired false positive rate and to the estimated set sizes to index.

4.2.2 K-mer aggregative methods summary

◇ **Sequence Bloom trees** SBTs are a family of related techniques detailed across multiple publications (8, 46–48). They are an adaptation of a hierarchical index structure of BFs that was initially developed outside of bioinformatics (49). At a high level, an SBT method builds a binary search tree (see Box 3 (a) for more details), whose leaves correspond to datasets in $\mathcal{D}$ and each internal node u represents the subset of $\mathcal{D}$ which appears as descendants of u (lets call these $desc(u) \subseteq \mathcal{D}$). A set query Q starts at the root and propagates down the whole tree. At any node, the information stored at that node is used to determine which k-mers of Q have identical presence and/or absence status in all $desc(u)$. These k-mers are then said to be determined and removed from Q as it propagates down the tree (see Box 3). When enough k-mers become determined, the search can be pruned (i.e. not propagated further down the tree).
Where the SBT methods differ is, to a large extent, in how the information at each node is stored. The initial SBT approach stored the BF of the corresponding dataset at each leaf and, at an internal node u, a BF of the union of $desc(u)$'s k-mers (8). Three later works brought optimizations to the original SBT concept. The first (simultaneously discovered in (46, 47)) was for each node to store two BFs, one containing the k-mers present in all $desc(u)$, and the other containing k-mers absent from all $desc(u)$. Split-Sequence Bloom trees (SSBT) (46) authors also noticed that once a k-mer is stored in a BF of an internal node u, it can be removed from the BFs of u's subtree to reduce space. AllSome SBTs (47) also showed that improving the tree topology through hierarchical clustering results in earlier pruning/faster queries, and lower space usage. HowDe SBTs (48) extend the last two works by further optimizing the way the information is stored at the two nodes, allowing a non-binary tree topology, and providing the first analytical analysis of the running time and memory usage of the various SBT approaches. All these improvements (46–48) greatly reduced the space and query time compared to the original SBT approach. (see Supplementary Box 9 for a comparison of the three SBT improvements).

◇ **BIGSI/COBS/DREAM-Yara.** A radically different approach was proposed in BIGSI (12) (Box 3 (b)). The BFs are stored in a matrix, with each column corresponding to dataset. It is useful to imagine this as the color matrix from previous sections, with the rows corresponding to the k-mers in $\mathcal{D}$[1]. Importantly, the matrix is stored in row-major order, i.e. row-by-row, so that each row appears as a consecutive block. Thus, a k-mer query extracts that k-mers presence or absence bit vector in $\mathcal{D}$ (see Box 3). The set query Q is then answered by extracting the bit vector for each k-mer in Q and performing bitwise operations on these vectors to answer the set query. A closely related work is presented as a part of DREAM-Yara (50), where BFs are interleaved, in order to efficiently retrieve the same position of several BFs (see Appendix Box 9).
The false positive rate of a BF is monotonically increasing in m/n, where m is the number of k-mers in the dataset and n is the number of bits in the BF. BIGSI uses the same size n for all the BFs, thus the false positive rates of the BFs differ depending on how many k-mers are in the corresponding dataset. The idea behind COBS (45) is that the size of each

[1] This is not technically true, since BFs are not exactly presence or absence bit vectors, but it captures the intuition.

BF (i.e. n) can be customized to its corresponding dataset, optimizing space usage while maintaining a constant false positive rate across datasets. COBS bins the datasets according to their estimated cardinality, and then uses a different BF size for each bin. COBS also improves the implementation of BIGSI and adds support for parallelization. Another advantage of COBS is that the index does not need to be fully loaded into RAM to perform queries, a feature that is also present in SBTs. Indeed, each bin can be loaded from disk and queried separately. Detailed examples can be found in Supplementary Box 7.

◇ **RAMBO** Similarly to BIGSI and COBS, RAMBO stores k-mers presence or absence using several BFs in a matrix. As in SBTs, the goal is to reduce the query complexity to be sublinear in the number of datasets. But the approach radically differs from SBTs as RAMBO does not create a hierarchy of union BFs in a tree. Also, instead of simply concatenating BFs as columns of the matrix, RAMBO defines each column to be a combination (in fact, a union) of multiple BFs, corresponding to multiple k-mer sets. Since each cell now represents information from different k-mer sets (i.e., here, the union of their BFs), some redundancy is necessary in the structure so that the original k-mer sets can be recovered. More details on the algorithm are provided in Appendix and Supplementary Box 9.

4.2.3 Additional aspects.

4.2.4 Compression. The different flavors of SBTs optionally compress BFs in the intermediate and leaf nodes of the tree using bit vector compression (typically RRR). To our knowledge BIGSI, COBS and RAMBO make no use of compression.

◇ **Insertion of new datasets.** In SBTs, the insertion of a new dataset requires computing a new BF for the sample and adding this BF as a leaf to the tree. Then all the BFs on the path from that leaf to the root are updated. In BIGSI, the BF is stacked into the matrix. The COBS implementation currently does not support insertion without rebuilding the complete index.

◇ **DBG traversal.** Unlike color-aggregative methods, we note here that k-mer-aggregative methods do not support DBG traversal.

4.3 Other schemes. Other unpublished tools have considered different techniques for storings and indexing sets of k-mer sets. **kamix**[2] uses samtools's BGZF compression library to store and index a k-mer matrix. From the same author, **kad**[3] uses a RocksDB database to store a list of k-mer and counts.

◇ **Schemes based on BWT.** BEETL (51) is a technique that stores inside a BWT all sequences (i.e., not k-mers, but the original data) from a sequencing dataset. BEETL was able to compress and index 135 GB of raw sequencing data into a 8.2 GB space (on disk for storage, or in memory for queries). A variant, BEETL-fastq (52), also enabled to perform efficient queries and was also applied to the representation of multiple datasets.

Population BWT (53) is also a scheme based on BWT geared towards the indexing of thousands of raw sequencing datasets. The BWT allows to query k-mers of any length and additionally gives access to the position of each k-mer occurrence within the original reads (note however that the raw reads have been error-corrected).

Section 5: Performance overview

5.1 Index construction on human RNA-seq samples Indexing datasets of a similar type, such as RNA-seq samples from a given species, was one of the first applications proposed in the literature of sets of k-mer sets, and remains one of the main benchmarks for these tools. Table S4 reports the performance of most of the recent tools on a collection of human RNA-seq datasets (2,652 RNA-seqs from the original SBT article[4]). This table was extrapolated by gathering results from three recent articles (11, 12, 48). As the articles use different hardware and slightly different parameters, a direct comparison of the tools is challenging. Instead, Table 1 presents a summary of the best possible performance that can be currently achieved on the given datasets.

The data processing phase, where the k-mer sets are constructed and initialized, often takes significant time across all methods. It is usually not presented as a bottleneck since it is viewed that this step can be computed while downloading the samples. Regarding query times, each method had used different experimental setups, making comparisons difficult, e.g., using transcript batches of different sizes (100-10,000). We refer the reader to the experimental benchmark in (48), which compares the average query times for randomly selected batches, effects of warming the cache, maximum peak memory for queries. Finally, we note that the information output by a query can vary from one implementation to another (see Table S2) and that the maximum supported value for k-mer size is also implementation-dependent.

5.2 Indexing bacterial genomes We now turn to the indexing of large collections of bacterial datasets. Table S5 summarizes benchmarks published in the article of COBS (45) and Vari-Merge (16). The evaluated tools are SBT, SSBT, AllSomeSBT, HowDeSBT, BIGSI, COBS, Vari, Vari-Merge, Rainbowfish, BFT, Multi-BRWT and Mantis+MST. Again we report here the best performing tool for each metric. We note that the datasets used to evaluate COBS and Vari-merge are different microbial collections, and thus, the presented results were extrapolated from different publications (see Table S5 caption for details). Again, the more recent methods tend to perform better at all levels. BIGSI, AllSomeSBT, and COBS queries are notably fast. SSBT

[2] https://github.com/jaudoux/kamix
[3] https://github.com/jaudoux/kad

[4] https://www.cs.cmu.edu/~ckingsf/software/bloomtree/srr-list.txt

Box 3. Techniques for k-mer aggregation.

Indices

In k-mer aggregative strategies, Bloom filters representing each of the initial datasets are organized in either a tree or a matrix structure. In the example below, there are four datasets (red, blue, green, and yellow), and the grey rectangles represent Bloom filters. Black bars in the BF represent the presence of k-mers.

(a) Tree strategy A search tree is constructed, where each leaf is a dataset and internal nodes represent groups of datasets. Datasets with similar BFs can be clustered to reside in the same subtree. In the original SBT approach (8), each node stores exactly one BF, containing all the k-mers present in the datasets of its subtree. For a leaf, this is simply the k-mers in the corresponding dataset. Later versions of SBTs (46–48) store more sophisticated data at each node, though they still rely on BFs.

(b) Matrix strategy The BFs from all the datasets are concatenated column-wise to obtain a matrix. A row in the matrix roughly corresponds to a k-mer (more precisely, to the position indicated by the hash value of a k-mer). In the original BIGSI approach (12), all BFs have exactly the same size. In COBS (45), datasets of comparable cardinality are grouped into bins, leading to a collection of matrices of different sizes.

Queries

(a) Tree strategy A query is one or more k-mers. Conceptually, one starts at the root node and then explores down the tree, always checking all the children of a node before moving to another node (breadth-first strategy). A counter of absent k-mers is maintained for each node. If it exceeds a certain threshold, the search does not propagate further down the subtree of that node.

(b) Matrix Strategy In BIGSI, each k-mer corresponds to a single BF location. The corresponding rows in the matrix are then extracted, and summed column by column to obtain a vector where each element contains the number of k-mers occurring in the corresponding dataset.

and COBS also have low RAM consumption. An advantage of HowDeSBT is the small size of the index on disk. This demonstrates that highly-diverse datasets, in terms of k-mer contents, can also be efficiently stored in variants of SBTs.

5.3 Indexing human genome sequencing data To the best of our knowledge, only two methods (BEETL-Fastq and Population BWT) have been applied to the representation of full read information from cohorts of whole human genomes. BEETL-Fastq represented 6.6 TB of human reads in FASTQ format in 1.7 TB of indexed files. Population BWT managed to index (in a lossy way) 87 Tbp of data, corresponding to 922 billion reads from the 1,000 genome project. After read correction and trimming, the authors obtained a set of 53 billion distinct reads (4.9 Tbp) and indexed it with a BWT stored with 464 GB on disk (requiring 561 GB of main memory for query). Metadata (e.g., sample information for each read) was stored in a 4.75 TB database.

Given the apparent difficulty to perform large-scale experiments on human datasets, we conclude that this is not yet a mature operation. Therefore did not provide a detailed comparison with RNA-seq and bacterial indexing techniques.

Section 6: Discussion

General observations can be derived from the comparison we have presented in this survey.

SBT approaches were designed for collections with high k-mer redundancy, such as human RNA-seq. In contrast, BIGSI and COBS focused on indexing heterogenous k-mer sets, such as k-mers originating from various bacteria. However, HowDeSBT demonstrated that SBTs could also perform well on this type of data. A trade-off exists between the construction time – in favor of BIGSI – and index size – in favor of the SBT. As shown in COBS paper, the BF resizing allows to gain memory, but the latest SBT techniques also have a lightweight memory footprint. Moreover, smaller BFs increased the false positive rate of COBS in comparison to other BF-based techniques (45).

It is also important to note that, across a number of methods, queries are approximate, although a number of color-aggregative methods support exact queries. Some cDBG im-

Data set	Data Processing Time (days)	Max Ext. Memory (GB)	Time (h, wallclock)	Peak RAM (GB)	Index Size (GB)
human RNA-seq (2,652 datasets)	2.5 (48) (HowDeSBT)	30 (48) (HowDeSBT)	2 (11) (SeqOthello)	5 (11) (SSBT)	15 (48) (HowDeSBT)
bacterial genomes (4,000 datasets)	Not reported	12 (45) (COBS)	0.05 (45) (COBS)	6.1 (45) (SSBT)	7.6 (45) (HowDeSBT)

Table 1. Overview of best-performing tools in terms of space and time requirements to build indices. These results are extrapolated from several publications. BIGSI and VARI-Merge results were also included during evaluations. The data processing time column refers to the time necessary to convert the original sequence files to the k-mer set indices (Bloom filters / CQF / Othello). The maximum external memory column corresponds to the peak disk usage when building the index. The time column is the time required to build the set of k-mer sets index (on one processor). The index size column is the final index size.

plementations (Vari and Vari-Merge) support additional features such as bubble-calling and graph traversal. New query types should also be considered. For instance, the possibility to obtain the k-mer counts instead of presence or absence would greatly assist in gene expression studies.

Color-aggregative methods and BIGSI/COBS seem better suited to query large sequences. Indeed, in these methods, the bottleneck for a single query is loading the index into memory. Then, the rest consists of k-mer hashing, roughly in constant time per each k-mer. Henceforward, once the index is loaded in memory, batches of queries or large queries can be answered very rapidly. Query speed can be improved depending on the implementation. For instance, in some data-structures such as the CQF in Mantis, consecutive k-mers are likely to appear nearby in memory, thus reducing the number of cache misses during a query. A drawback is that these structures are usually more memory consuming than SBTs. Moreover, in the case of SBTs, BIGSI and COBS, large queries allow mitigating the underlying BF false positive rate. SBTs and COBS do not need to load the entirety of the index into memory at query time since the query iteratively prunes irrelevant datasets. That is why SBTs are more suitable for short queries, while for large queries, k-mer look-ups become a bottleneck. For very large queries (e.g. the k-mers from a whole sequencing experiment), only AllSomeSBT (47) has an efficient specialized algorithm.

Moreover, the fraction of k-mers matching a set query, which is mainly used as a similarity proxy in those works, could be further explored from a biological point of view. For instance, a single substitution in a base is covered by k different k-mers. If the indexed sequences differ from the query on that substitution, these k-mer will not be found in the structure, and the match could be missed. The effect of changing the size of k has not been sufficiently assessed, nor has the biological impact on the results been evaluated when tuning the θ parameter (with the exception of RNA-seq quantification (8)).

While there have been extensive empirical benchmarks to compare the performance of the different methods, analytical comparisons of their performance has been limited (see (48) for an example, though it is limited to only intra-SBT comparisons). The difficulty in using worst-case analysis to analyze performance in this case is that the methods are really designed to exploit the properties of real collections, and worst-case analysis is therefore not helpful. Progress can be made by coming up with appropriate models to capture the essential properties of real data and analyzing the methods using those models.

The data structures surveyed in this paper should be seen as initial attempts from the community toward being able to routinely query the hundreds of thousands of samples deposited in public repositories (e.g., SRA) or private ones. An essential next step would be to have user-friendly tools. User friendliness can be seen from different perspectives. First, one may try to cast more concrete biological questions into simplified k-mer queries that can then be asked to the indices. Second, the results of queries could be presented in a manner that is more suitable to biologists rather than their current form, consisting mainly of the output of k-mer queries. For instance, a list of reads contained in the indexed datasets could be output for further investigation. However, indexing reads is more challenging, and this direction would require new developments for the structures to scale. Third, special attention given to user interfaces could help broaden the usage of these methods. Web interfaces[5] are challenging to maintain in the long run; thus another solution could be to provide offline pre-computed indices. This way, users would only download some chunks of interest from the index for further investigation.

ACKNOWLEDGEMENTS

This work was supported by ANR Transipedia (ANR-18-CE45-0020) and INCEPTION (PIA/ANR-16-CONV-0005). This material is based upon work supported by the National Science Foundation under Grants No. 1453527 and 1439057 to PM. The authors are grateful to Jan Holub for feedback on the BOSS section, and to Daniel Gautheret for his comments and corrections.

AUTHOR CONTRIBUTIONS

COMPETING FINANCIAL INTERESTS

The authors declare no competing financial interests.

Section 7: Bibliography

1. Laura Clarke, Xiangqun Zheng-Bradley, Richard Smith, Eugene Kulesha, Chunlin Xiao, Iliana Toneva, Brendan Vaughan, Don Preuss, Rasko Leinonen, Martin Shumway, et al. The 1000 genomes project: data management and community access. Nature methods, 9(5): 459, 2012.
2. Clare Turnbull, Richard H Scott, Ellen Thomas, Louise Jones, Nirupa Murugaesu, Freya Boardman Pretty, Dina Halai, Emma Baple, Clare Craig, Angela Hamblin, et al. The 100 000 genomes project: bringing whole genome sequencing to the nhs. BMJ, 361:k1687, 2018.
3. Tuuli Lappalainen, Michael Sammeth, Marc R Friedländer, Peter AC't Hoen, Jean Monlong, Manuel A Rivas, Mar Gonzalez-Porta, Natalja Kurbatova, Thasso Griebel, Pedro G Ferreira, et al. Transcriptome and genome sequencing uncovers functional variation in humans. Nature, 501(7468):506, 2013.
4. Ruth E Timme, Hugh Rand, Maria Sanchez Leon, Maria Hoffmann, Errol Strain, Marc Allard, Dwayne Roberson, and Joseph D Baugher. Genometrakr proficiency testing for foodborne pathogen surveillance: an exercise from 2015. Microbial genomics, 4(7), 2018.
5. The MetaSUB International Consortium. The metagenomics and metadesign of the subways and urban biomes (metasub) international consortium inaugural meeting report. Microbiome, 4(1), 2016. doi: 10.1186/s40168-016-0168-z.
6. Charles E Cook, Rodrigo Lopez, Oana Stroe, Guy Cochrane, Cath Brooksbank, Ewan Birney, and Rolf Apweiler. The european bioinformatics institute in 2018: tools, infrastructure and training. Nucleic acids research, 47(D1):D15–D22, 2018.

[5]The group maintaining BIGSI proposed one: http://www.bigsi.io/

7. Rasko Leinonen, Hideaki Sugawara, Martin Shumway, and International Nucleotide Sequence Database Collaboration. The sequence read archive. Nucleic acids research, 39 (suppl_1):D19–D21, 2010.
8. B. Solomon and C. Kingsford. Fast search of thousands of short-read sequencing experiments. Nature Biotechnology, 34(3):300–302, 2016.
9. Sara A Byron, Kendall R Van Keuren-Jensen, David M Engelthaler, John D Carpten, and David W Craig. Translating rna sequencing into clinical diagnostics: opportunities and challenges. Nature Reviews Genetics, 17(5):257, 2016.
10. Katarzyna Tomczak, Patrycja Czerwińska, and Maciej Wiznerowicz. The cancer genome atlas (tcga): an immeasurable source of knowledge. Contemporary oncology, 19(1A):A68, 2015.
11. Y. Yu, J. Liu, X. Liu, Y. Zhang, E. Magner, E. Lehnert, C. Qian, and J. Liu. Seqothello: querying rna-seq experiments at scale. Genome Biology, 19(1):167, 2018.
12. Phelim Bradley, Henk C den Bakker, Eduardo PC Rocha, Gil McVean, and Zamin Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. Nature biotechnology, 37 (2):152, 2019.
13. Alexandre Almeida, Stephen Nayfach, Miguel Boland, Francesco Strozzi, Martin Beracochea, Zhou Jason Shi, Katherine S Pollard, Donovan H Parks, Philip Hugenholtz, Nicola Segata, et al. A unified sequence catalogue of over 280,000 genomes obtained from the human gut microbiome. bioRxiv, page 762682, 2019.
14. Grace Blackwell, Zamin Iqbal, and Nick Thomson. Evolution and spread of bacterial transposons. Access Microbiology, 1(1A), 2019.
15. Elizabeth A Miller, Ehud Elnekave, Cristian Flores-Figueroa, Abigail Johnson, Ashley Kearney, Jeannette Munoz-Aguayo, Kaitlin A Tagg, Lorelee Tschetter, Bonnie Weber, Celine Nadon, et al. Emergence of a novel salmonella enterica serotype reading clone is linked to its expansion in commercial turkey production, resulting in unanticipated human illness in north america. bioRxiv, page 855734, 2019.
16. Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de Bruijn graphs via merging. Bioinformatics, 35(14):i51–i60, 07 2019. ISSN 1367-4803. doi: 10.1093/bioinformatics/btz350.
17. Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent sets of k-long dna sequences. CoRR, abs/1903.12312, 2019.
18. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, 1970.
19. Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. Journal of computational biology, 19(5):455–477, 2012.
20. Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. Nature genetics, 44(2):226, 2012.
21. Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. PVLDB, 5(11):1627–1637, 2012.
22. Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. A general-purpose counting filter: Making every bit count. In Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, pages 775–787. ACM, 2017.
23. Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An efficient, scalable and exact representation of high-dimensional color information enabled via de bruijn graph search. In International Conference on Research in Computational Molecular Biology, pages 1–18. Springer, 2019.
24. Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul S. Morley, Keith E. Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher. Succinct colored de bruijn graphs. Bioinformatics, 33(20):3181–3187, 2017.
25. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. J. ACM, 52(4):552–581, 2005.
26. Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings, volume 7534 of Lecture Notes in Computer Science, pages 225–235. Springer, 2012.
27. Christina Boucher, Alex Bowe, Travis Gagie, Simon J Puglisi, and Kunihiko Sadakane. Variable-order de bruijn graphs. In 2015 Data Compression Conference, pages 383–392. IEEE, 2015.
28. Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms for Molecular Biology, 8(1):22, 2013.
29. Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA., pages 233–242. ACM/SIAM, 2002.
30. R. M. Fano. On the number of bits required to implement an associative memory. Memorandum 61, Computer Structures Group, MIT, Cambridge, MA, 1971.
31. P. Elias. Efficient storage and retrieval by content and address of static files. Journal of the ACM, 21(2):246–260, 1974.
32. Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In Proc. 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014, pages 273–282. ACM, 2014.
33. Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with roaring. Software: Practice and Experience, 46(11):1547–1569, 2016.
34. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, pages 841–850. Society for Industrial and Applied Mathematics, 2003.
35. Mikhail Karasikov, Harun Mustafa, Amir Joudaki, Sara Javadzadeh-No, Gunnar Rätsch, and André Kahles. Sparse binary relation representations for genome graph annotation. In International Conference on Research in Computational Molecular Biology, pages 120–135. Springer, 2019.
36. Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. Algorithms for Molecular Biology, 11 (1):3, 2016.
37. Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: A succinct colored de bruijn graph representation. In 17th International Workshop on Algorithms in Bioinformatics (WABI 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
38. Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: a fast, small, and exact large-scale sequence-search index. Cell systems, 7(2):201–207, 2018.
39. Harun Mustafa, Ingo Schilken, Mikhail Karasikov, Carsten Eickhoff, Gunnar Rätsch, and André Kahles. Dynamic compression schemes for graph coloring. Bioinformatics, 35(3): 407–414, 2018.
40. Guillaume Holley and Páll Melsted. Bifrost–highly parallel construction and indexing of colored and compacted de bruijn graphs. BioRxiv, page 695338, 2019.
41. Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. Bioinformatics, 34(13):i169–i177, 06 2018.
42. Camille Marchet, Mael Kerbiriou, and Antoine Limasset. Indexing de bruijn graphs with minimizers. bioRxiv, 2019. doi: 10.1101/546309.
43. Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinformatics, 32(12):i201–i208, 06 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw279.
44. Ilia Minkin, Son Pham, and Paul Medvedev. Twopaco: An efficient algorithm to build the compacted de bruijn graph from many complete genomes. Bioinformatics, 33(24):4024–4032, 2016.
45. Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. Cobs: a compact bit-sliced signature index. arXiv preprint arXiv:1905.09624, 2019.
46. B. Solomon and C. Kingsford. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. Journal of Computational Biology, 25(7):755–765, 2018.
47. Chen Sun, Robert S. Harris, Rayan Chikhi, and Paul Medvedev. Allsome sequence bloom trees. In Research in Computational Molecular Biology - 21st Annual International Conference, RECOMB 2017, Hong Kong, China, May 3-7, 2017, Proceedings, volume 10229 of Lecture Notes in Computer Science, pages 272–286, 2017.
48. R. S. Harris and P. Medvedev. Improved representation of sequence bloom trees. bioRxiv, 2018. doi: https://doi.org/10.1101/501452.
49. Adina Crainiceanu and Daniel Lemire. Bloofi: Multidimensional bloom filters. Information Systems, 54:311–324, 2015.
50. Temesgen Hailemariam Dadi, Enrico Siragusa, Vitor C Piro, Andreas Andrusch, Enrico Seiler, Bernhard Y Renard, and Knut Reinert. Dream-yara: An exact read mapper for very large databases with short update time. Bioinformatics, 34(17):i766–i772, 2018.
51. Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. Bioinformatics, 28(11):1415–1419, 05 2012. ISSN 1367-4803. doi: 10.1093/bioinformatics/bts173.
52. Lilian Janin, Ole Schulz-Trieglaff, and Anthony J Cox. Beetl-fastq: a searchable compressed archive for dna reads. Bioinformatics, 30(19):2796–2801, 2014.
53. Dirk D Dolle, Zhicheng Liu, Matthew Cotten, Jared T Simpson, Zamin Iqbal, Richard Durbin, Shane A McCarthy, and Thomas M Keane. Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes. Genome research, 27(2):300–309, 2017.
54. Roberto Grossi and Giuseppe Ottaviano. The wavelet trie: Maintaining an indexed sequence of strings in compressed space, 2012.
55. Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. arXiv preprint arXiv:1702.03154, 2017.

Section 8: Appendix.

Appendix outline The Appendix is divided in two parts, namely, details on methods and details on the benchmark. Explanations are given in the main text, and we provide concrete examples in Supplementary Boxes. In the Methods Section, first, some useful definitions of a few computer science objects useful to this study are recalled. Then, details on relevant k-mer structures (hash-based, BWT-based) and compression are given. Finally, we give more insight about some of the set of k-mer sets approaches. In particular, we provide a lower-level description of structure/features that did not appear in the main document. E.g., an example BFT and RAMBO structures are given, as well as comparisons between specific approaches. Complexities are outlined in Table S1.

8.1 Method details.

8.1.1 k-mer index data structures.

Hash-based methods Bloom filters (See Supp. Box 1) Bloom filters are used to perform approximate membership on sets. They can be defined as a binary vector of size n bits and a set of l hash functions $h_0,\ldots,h_{l-1}$. An element is inserted in the filter by hashing it using the l hash functions and by setting the corresponding bits to 1. The query follows straightforwardly, requiring only to check if all bits corresponding to the query are 1 values, even in case of collision. BFs can lead to return false positives but never false negative (i.e. they are always able to retrieve a member of the indexed set). Depending on the filter size n and the number of hash functions l, a trade-off on the false positive rate can be found.

Counting Quotient filter intuition (See Supp Box 1) A CQF is another kind of filter that uses a different hashing strategy. It is based on the quotient filter notions. The quotient filter uses two hashes of the element (remainder and quotient) to identify elements in a table. The *quotient* indicates the position where the element information is stored, and the *remainder* is written at this position. Collisions are managed by using linear probing, and the filter can be exact if revertible hash functions are used. This QF can be modified to store a count information associated to each element.

Othello hashing Othello hashing allows to store (approximately) the association between elements and the set they belong to ((8) in Othello Hashing Figure of Supp. Box 1). For instance with two initial sets S_1 and S_2, the key idea is that any element from S_1 will be associated to two different values stored at positions defined using two hash functions, while any element from S_2 will be associated to two identical values. When querying, an element is hashed twice and we simply verify whether values of the associated pair are identical or not (alien keys can yield false positives). Othello hashing can be generalized to more than two input sets in our application case. Hash collisions when inserting new keys imply cascading modifications to other pairs of values (see (4) and (5) in Othello Hashing Figure of Supp. Box 1). In practice, the method uses a graph representation to propagate modifications (sometimes insertions yield cyclic patterns, in this case, elements are stored apart in another secondary structure).

Graphs and trees A graph is a pair of two sets V and E ((a) in Figure S1). Elements of V are nodes and pairs of related nodes are called edges, which are elements of E.
A path in a graph is a sequence of edges that joints a sequence of distinct nodes.
A tree is a particular graph in which any two nodes are connected by exactly one path ((b) in Figure S1). SBT relies on trees. A forest is a disjoint union of trees. HowDeSBT's tree simplifications can lead to a forest. A subtree is a subset G' and E' of a tree $T=(G,E)$. Queries in SBT are performed by descending in subtrees.
A trie is a tree that allows to efficiently save and query a set of words. Wavelet tries and Bloom filter tries (BFT) are particular kind of tries. Wavelet tries (54), used in Metannot, are designed to store compressed sequences. Details on BFT are given in Supp. Box 3.

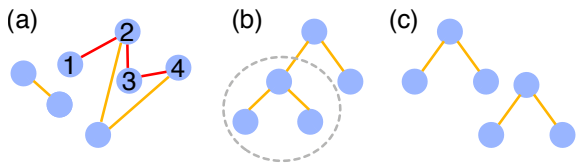


Fig. S1. (a) A graph. Nodes pictured in blue, edges in orange and red. A path is drawn between nodes 1,2,3,4 through red edges. (b) A tree, and one of its subtrees is circle in grey. (c) A forest.

De Bruijn graphs were defined in the main document. Compacted De Bruijn graphs derive from their definition. Let p be a path in the DBG, i.e., p is a set of n connected vertices $p=\{x_1,x_2,\ldots,x_n\}$. If p is not a cycle, if x_1 has more than one ingoing edges and only one outgoing edge, if x_n has more than one outgoing edges and only one ingoing edge, and if all other edges

have one and only one outgoing and ingoing edges, then the path p can be compacted. This means that the set of vertices can be fused in an single vertex, containing the sequence of all k-mers assembled, following the order of the directed path. These paths represent consecutive k-mers with no ambiguity. A DBG in which all possible paths have been compacted is called a compacted DBG. Compacted DBGs have the advantage to use generally less space to represent the same amount of nucleotides, than a regular DBG (see Supp. Box 2).

BOSS: BWT-based De Bruijn Graphs The Burrows Wheeler Transform (BWT) is a text transformation algorithm. It receives a sequence as input, and rearranges its characters in a way that enhances further compression. The transformation is reversible, thus the original sequence can be decoded. BOSS rearranges k-mers in the De Bruijn graph in a similar way.
Here, we briefly show how the BOSS scheme works. To begin we describe the following simple — but not space-efficient — representation of a DBG: take each unique $(k+1)$-mer, consisting of a vertex concatenated to the label of an outgoing edge, and sort those $(k+1)$-mers according to their first k symbols taken in reverse order. The resulting sorted list contains all nodes and their adjacent edges sorted such that all outgoing and incoming nodes of a given node can be identified. Thus, it is a working representation, in that all graph operations can be performed, but is far from space efficient since $(k+1)$ symbols need to be stored for each edge. Next, we show that we can essentially ignore the first k symbols, which will lead to a substantial reduction in the total size of the data structure.
First, we make a small alteration to this simple representation by padding the graph to ensure every vertex has an incoming path made of at least k vertices, as well as an outgoing edge. This maintains the fact that a vertex is defined by its previous k edges. For example, say k-mer `CCATA` has no incoming edge; then we add a vertex `$CCAT` and an edge between `$CCAT` and `CCATA`, then between `$$CCA` and `$CCAT`, and so forth. We let W be the last column of the sorted list of $(k+1)$-mers. Next, we flag some of the edges in the representation with a minus symbols to disambiguate edges incoming into the same vertex – which we accomplish by adding a minus symbol to the corresponding symbols in W. Hence, W is a vector of symbols from {`A, C, G, T, $, -A, -C, -G, -T, -$`}. Next, we add a bit vector L which represents whether an edge is the last edge, in W, exiting a given vertex. This means that each node will have a sequence of zero-or-more 0-bits followed by a single 1-bit, e.g., if there is only a single edge outgoing from a node then there is a single 1-bit for that edge. Overall the representation consists of a vector of symbols (W), a bit vector (L) implemented using a rank/select (29) data structure, and finally an array that records the counts of each character. It may seem surprising but these three vectors provide enough information for representing the DBG and supporting traversal operations. We refer the reader to the original paper for a detailed discussion. Lastly, we note that this representation, which is referred to as BOSS, is due to Bowe et al. (26) and was extended for storing colors (24) (see Supplementary Box 4 for an example).

8.1.2 Details on compression To efficiently represent a $n \times c$ color matrix, over n k-mers across c datasets, different schemes have been proposed. A color class is a set of colors. It can also be seen as a bit vector, and a line in the color matrix. Supp. Box 5 presents examples of the different techniques: the delta-based encoding used in Mantis+MST (c), the RRR/Elias-Fano coding (e) used e.g. in Mantis and VARI, the lossy compression using BF from Metannot (d), the BRWT principle (f), and the three strategies used in SeqOthello (g).

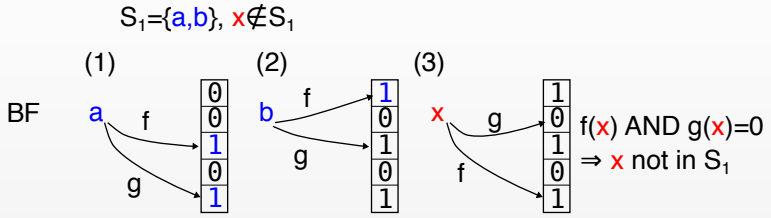
8.1.3 Set of k-mer sets details

Color aggregative methods. We first show how the different color aggregative methods combine k-mer sets and color strategies in Supp. Box 6. Then, we report the various hashing strategies used by methods that do not rely on BOSS in Supp Box 7.
BFT significantly differs from other methods: an example is shown in Supp. Box 3. In a BFT, k-mers are divided into a prefix and a suffix part that are recorded in tree (actually, a variant called a burst trie). Prefixes are further divided in chunks, which are inserted into the root and inner nodes of the tree. Suffixes are in the leaves. Queries start at the tree root and progress through the path that spell the query string. In practice, k-mer suffixes are stored as tuples with their corresponding color class. Bloom filters are also used in the inner nodes, to increase query speed by quickly checking the presence of a chunk. Note that the above description of BFT does not capture the full complexity of the data structure, and should only be used to build an initial intuition.

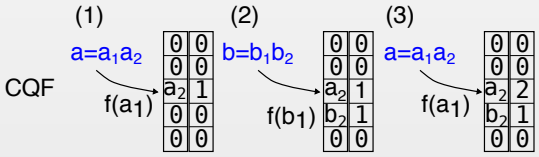
k-mer aggregative methods In Supp. Box 8, we present indexing and query details, in a similar fashion than Boxes 4 and 5 in the main text, but more in depth. We show the index construction and query steps in SBT, BIGSI, and shows how COBS improves on BIGSI's representation while keeping the core idea.
Then we illustrate current contrasts between k-mer-aggregative methods in Supp. Box 8. For the different SBTs works, different strategies are used to store information in each node. Supp. Box 8 shows the improvements in bit-vector representation first brought by SSBT/AllSomeSBT, then by HowDeSBT. In a second Figure, BIGSI, Dream Yara and RAMBO strategies for indexing Bloom filters are compared. In the following, we outline the very recent RAMBO's method.
An example of RAMBO structure is shown in Supp. Box 8, bottom right Figure. RAMBO builds a matrix of C columns and T rows. Cells of the matrix are BFs. At construction, a given dataset is assigned to one cell per column, and the corresponding

Supplementary Box 1. Hashing techniques

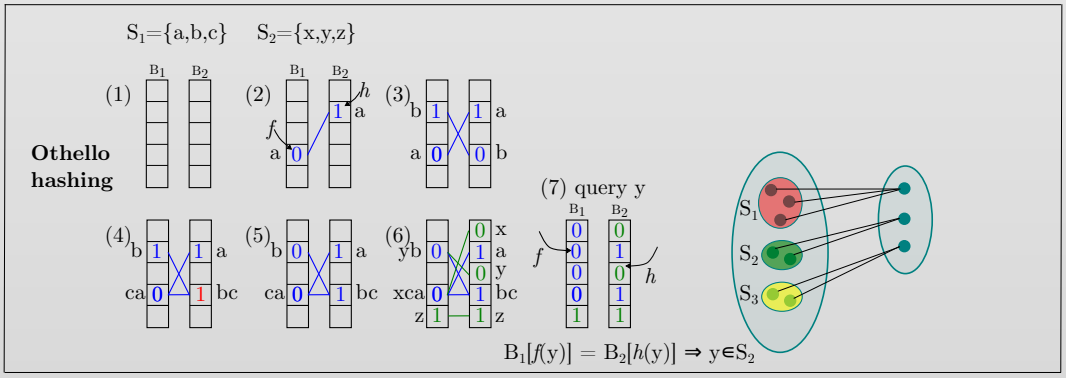
Bloom filters The example filter has a set of two functions f and g. In (1) a is inserted by putting 1s at positions 2 and 4 indicated by both functions. (2) b is inserted similarly. (3) x is queried, $g(x)$ giving a 0 we are certain that it is not present in the filter.



Counting Quotient filter intuition Element a and b are decomposed into a_1a_2 and b_1b_2. a_1, b_1 are quotients in the example, and a_2, b_2 are remainders. (1) During a's insertion, the quotient is used to find the position of the element in the filter, and a_2 is stored. The count is associated (second column). (2) similar operation for b. (3) a is re-inserted, leading to a count of 2.

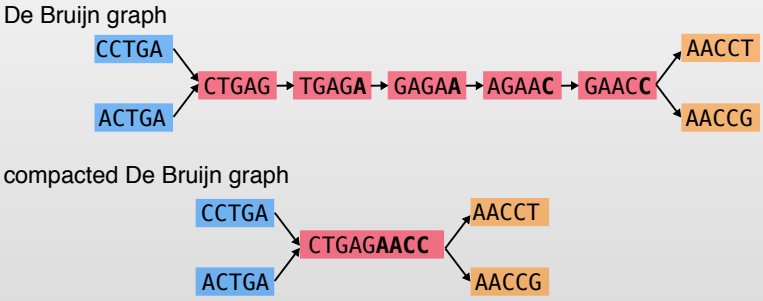


Othello Hashing intuition In the example below (figure), two sets S_1 and S_2 are hashed, but a larger number of sets can be dealt with. Othello hashing uses two hash functions, denoted here by f and h. The method maintains two bit arrays B_1 and B_2 (1). In (2), the element a from S_1 is hashed with f in B_1 and with h in B_2. A different value will be stored in B_1 and in B_2 (0 and 1). The lines between those two values visually represent their association to a. (3) b is hashed the same way than a, ensuring again that two different values are associated to b. (4) Element c is inserted, here we cannot ensure two different values are associated to c without having a contradiction. Thus b's 0 in B_2 is modified (in red). (5) The values associated to b must differ, so in B_1 we modify the 1 associated with b to a 0. (6) x, y, z are inserted, this time they must be associated to pairs of identical elements as they belong to S_2. (7) y is queried by hashing it with f and h and by checking if the associated values are identical (y in S_2) or different (y in S_1).



Supplementary Box 2. Compacted De Bruijn graph

In the example below, the first graph is a regular De Bruijn graph from the 5-mers CCTGA, ACTGA, CTGAG, TGAGA, GAGAA, AGAAC, GAACC, AACCT, AACG. CTGAG has two ingoing edges and only one outgoing, GAACC has two outgoing edges and only one in-going, any other vertex in-between connecting CTGAG and GAACC has only one in-going and outgoing edge. Thus this red path can be compacted.
The second graph is the resultant compacted De Bruijn graph. The red path becomes a single red node, by concatenating CTGAG, **A**, **A**, **C** and **C**. It keeps the same connections than the two flanking nodes. Each resultant node is referred to as a unitig.



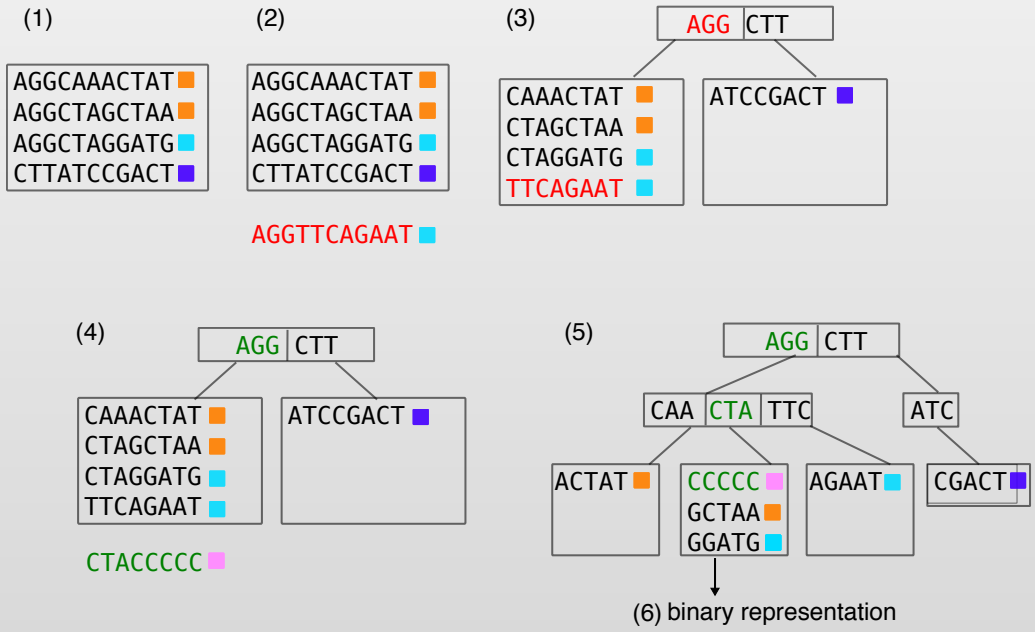
It is noticeable than instead of needing 5×9 nucleotides, this second representation only requires $5 \times 4 + 9$.

Supplementary Box 3. Bloom filter trie

A Bloom filter trie is a tree that stores sequences. Either a sequence of length k is stored in a leaf (which can store at most t sequences), or a node is "burst" (i.e. transformed) into a sub-tree. The new sub-tree consists of a node v and two or more children of v. All prefixes of length p from the sequences in the original leaf are stored in v. All suffixes of length $k-p$ that follow the i-th prefix in v are stored in the i-th child of v.

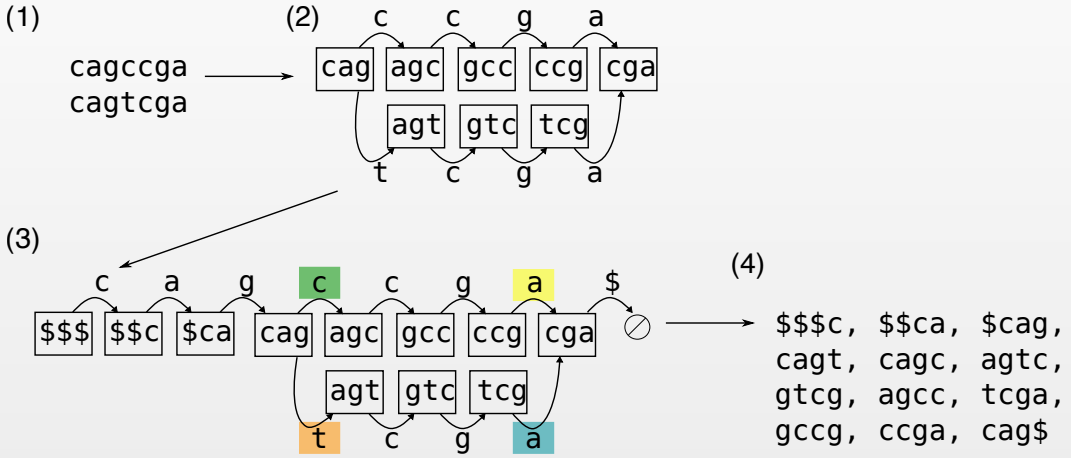
We now show how to store the following k-mers in a BFT: AGGCTAGCTAA, AGGCAAACTAT, AGGCTAGGATG, CTTATCCGACT, AGGTTCAGAAT, AGGCTACCCCC, with $t = 4$ and $p = 3$. (1) the first four k-mers can be inserted in a single leaf, since $t = 4$. (2) The fifth k-mer AGGTTCAGAAT (red) cannot be inserted in the leaf, requiring a burst operation. (3) To perform the burst, the prefixes of size p of the five k-mers are stored in the root. Each prefix has a pointer to its corresponding subtree. Suffixes of length $k-p$ are stored in the leaves. (4) AGGCTACCCCC (green) insertion should be made in the left leaf as its prefix is AGG. This requires a burst as the left leaf is full. (5) The burst operation is performed on this leaf and AGGCTACCCCC can be inserted. (6) Binary representation based on Bloom filters is used for the leaves. Note: k-mers are stored as tuples with their color class.

AGGCTAGCTAA,AGGCAAACTAT,AGGCTAGGATG,CTTATCCGACT,AGGTTCAGAAT,AGGCTACCCCC



Supplementary Box 4. BOSS graph structure

In this Box we describe the BOSS data structure, as per its original flavor ((26)). Let's assume we want to build the BOSS from the two sequences CAGCCGA and CAGTCGA with $k = 3$ in the figure below, part (1). Part (2) is the De Bruijn graph from these sequences (no reverse-complements are considered here). In this representation, each vertex contains a 3-mer, and an edge represents a 4-mer existing in the original sequences, the label of the edge being the last nucleotide of this 4-mer. (3) represents the same information, but with the constraint that any nodes not containing \$ must be preceded by k vertices (3 vertices) and must have at least an outgoing edge. Thus supplementary nodes are added. (4) is the list of $(k+1)$-mers in the (3) graph.



(5)

	nodes	edges labels
0	\$\$\$	c
1	\$ca	g
2	cga	\$
3	\$\$c	a
4	gcc	g
5	agc	c
6	gtc	g
7	cag	c
8	cag	t
9	ccg	a
10	tcg	a
11	agt	c

(6)

0	\$\$\$	c	1
1	\$ca	g	1
2	cga	\$	1
3	\$\$c	a	1
4	gcc	g	1
5	agc	c	1
6	gtc	g	1
7	cag	c	0
8	cag	t	1
9	ccg	a	1
10	tcg	a-	1
11	agt	c	1

(7)

0	\$\$**\$**	c	1
1	\$c**a**	g	1
2	cga	\$	1
3	\$\$**c**	a	1
4	gcc	g	1
5	agc	c	1
6	gtc	g	1
7	ca**g**	c	0
8	cag	t	1
9	ccg	a	1
10	tcg	a-	1
11	ag**t**	c	1

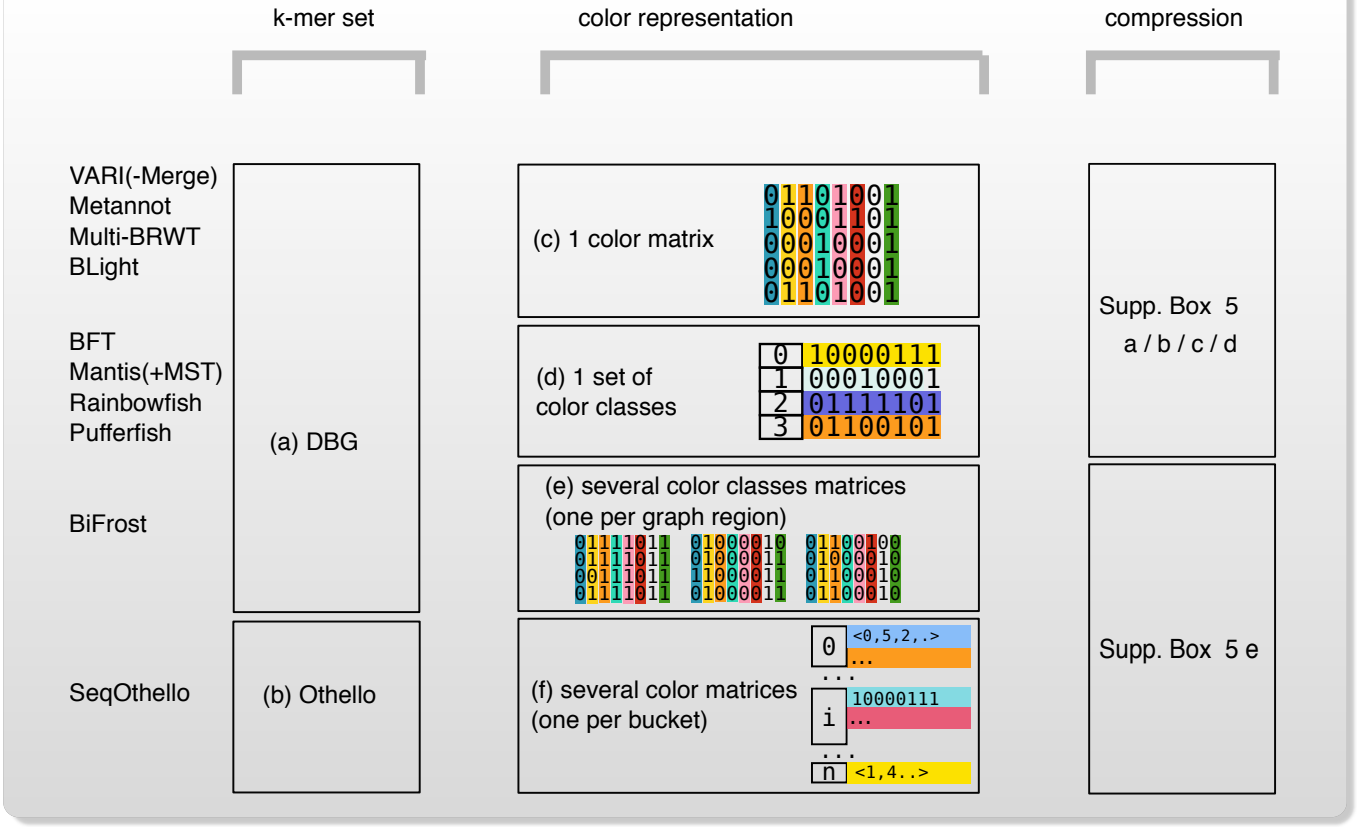
(8)

0	c	1
1	g	1
2	\$	1
3	a	1
4	g	1
5	c	1
6	g	1
7	c	0
8	t	1
9	a	1
10	a-	1
11	c	1

\$	**0**
a	**1**
c	**3**
g	**7**
t	**11**

(5) These $(k+1)$-mers are listed by lexicographic order by reading them in reverse from the kth nucleotide. This gives a matrix of nucleotides, the last nucleotide of each $(k+1)$-mer being in a separate red column. Each line of the matrix represents a node label in the graph, and the red vector represents the edge labels. (6) In order to denote nodes that have several outgoing edges, a new vector (blue) is used. 0s are put until the last edge of a node in the last, represented by a 1. Here node CAG has two edges labeled by C (green, marked 0) and T (orange, marked 1). When two nodes enter the same node, they have the same label, one is disambiguated using a $-$, as for yellow/blue labels. (7) Only the last column of the matrix will be kept in the BOSS. We retain the rank of each first symbol (in red): \$ appears at rank 0, A at rank 1, C at rank 3, ... (8) The final information in the BOSS structure. From these tables, DBG operations such as going forward, backward from a given node are shown to be possible in (26), but we do not describe them here.

Supplementary Box 5. Details on building blocks in color aggregative methods.



BF is updated with the presence of its k-mers. This creates the necessary redundancy in the structure. Since several datasets can be assigned to a same cell, BF become union BF by informing for the presence/absence of k-mers in more than one dataset. Query is performed on the rows, each union BF giving a row-wise union of sets where the query could be present. The final sets containing the query are deduced by intersection of the different set unions.

	construction	query
SBT	$\mathcal{O}(n \times b)^*$	$\mathcal{O}(Q \times h)$
VARI	$\mathcal{O}(N \times \log(N))$	$\mathcal{O}(Q \times n)$
Mantis	$\mathcal{O}(N \times n)$	$\mathcal{O}(Q \times n)$
SeqOthello	$\mathcal{O}(N \times n)$	$\mathcal{O}(Q \times n)$
BFT	$\mathcal{O}(N \times n)$	$\mathcal{O}(Q \times n)$
BIGSI	$\mathcal{O}(n \times b)$	$\mathcal{O}(Q \times n)$
RAMBO	unreported	$\mathcal{O}(Q \times \sqrt{n} \times log(n))$

Table S1. Time complexities for the construction and query of the main approaches. N is the total number of distinct k-mers, n is the number of datasets, Q the query size (number of k-mers). We denote by b the number of bits in a Bloom filter, and h the number of datasets that contain at least θ% of Q k-mers. We consider k as a relatively small constant (21-63). * This time is derived from Theorem 1 in (48) with the assumption that the size of the Bloom filter b is roughly $\mathcal{O}(N/n)$. Note that there may be an additional complexity cost for building the topology of the tree through clustering. We note that in the worst case (majority of k-mers present in all datasets), the query complexities of SBTs would be $\mathcal{O}(Q \times n)$.

Type of query output	Method
Datasets containing at least θ% of the queried k-mers	SBT, BIGSI, SeqOthello, Mantis, BFT
Fraction of query k-mers present in each experiment	SeqOthello
Bubble calling from query k-mers	VARI
Given a color, list of k-mers having that color	VARI-Merge

Table S2. Comparison of the different methods outputs.

	Graph	Color	Properties
Mantis	S	s,d	
Mantis+MST	S	S	
Metannot		S	Lossy
Multi-BRWT		S	
Rainbowfish	S	s	
VARI	S		
VARI-Merge	S	D	
BFT	S	s	
SeqOthello	N/A	S	Lossy
Bifrost	D	S,d	
Pufferfish	S		
BLight	S		

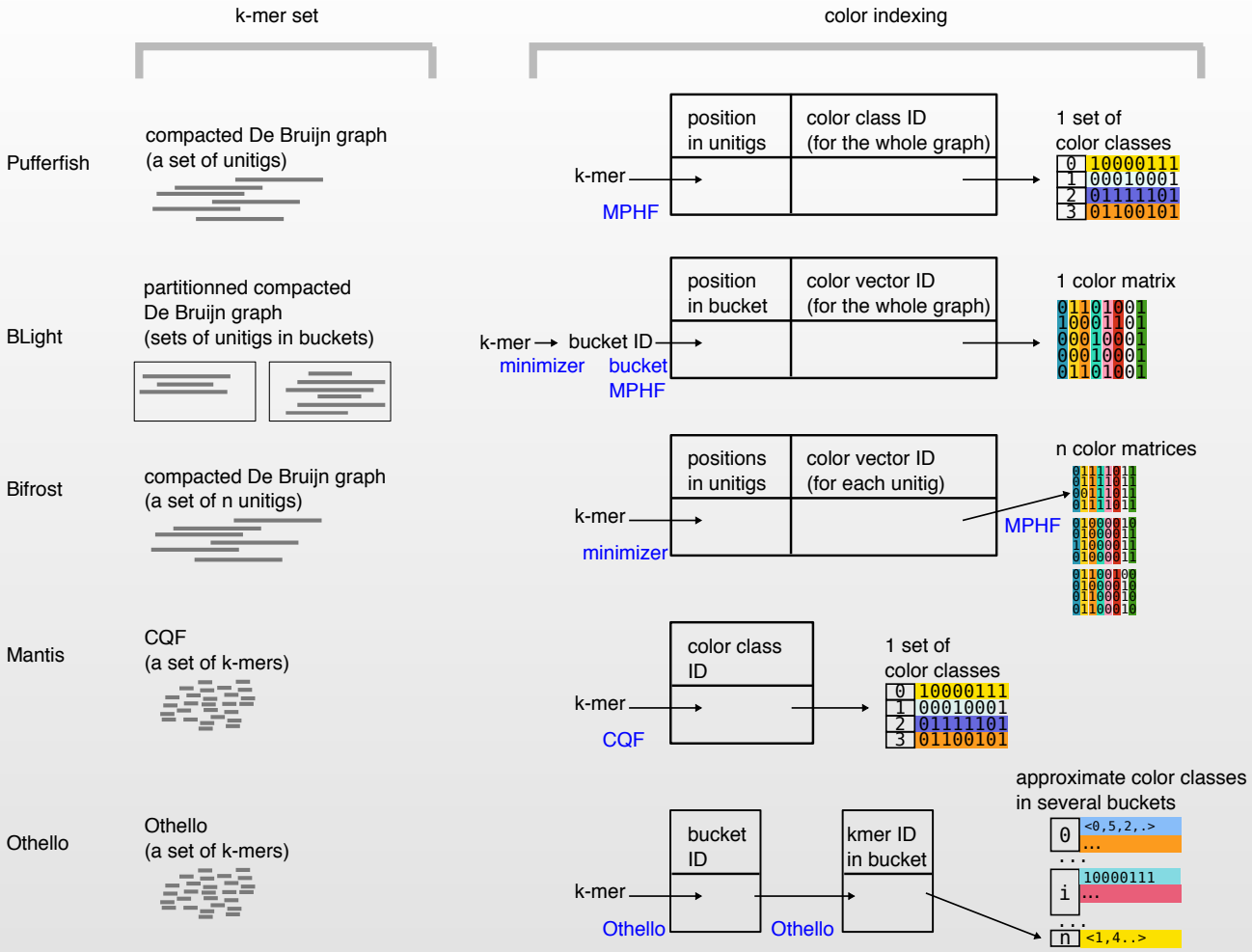
Table S3. Summary of color-aggregative methods's focus on scalability of their components. A capital 'S' indicates that the corresponding method claims to have focused on a highly scalable component for either its graph representation or its color representation. An 's' indicates moderately scalable, and "N/A" stands for non-applicable. Similarly, capital and regular 'D' and 'd' indicate a particular focus on the structure dynamicity. The tradeoff with lossy data-structures is indicated in the last column.

Tool	Data Processing Time (days)	Max Ext. Memory (GB)	Time (h, wallclock)	Peak RAM (GB)	Index Size (GB)
SBT	3.5^b	300^a	55^b	25^b	200^a
AllSomeSBT	3.5^a	600^a	25^a	35^b	140^a
SSBT	3.5^a	600^a	55^a	5^b	20^a
HowDeSBT	2.5^a	30^a	10^a	N/A	15^a
Mantis	130^a	3,500	20^a	N/A	30^a
SeqOthello	3.5^b	190^b	2^b	15^b	20^b
BIGSI	N/A	N/A	N/A	N/A	145^c

Table S4. Space and time requirements to build indices. The best result for each column is shown in green. ^{a} refer to HowDeSBT results, ^{b} refer to SeqOthello results, ^{c} refer to BIGSI results. The benchmark dataset has approximately 4 billion k-mers, within 66 files, where reads shorter than 20 bases are discarded. Despite using the same dataset, the studies vary slightly: they filter low frequency k-mers slightly differently (11, 38), and hardware used in experiments also differs. The data processing time column refers to the time necessary to convert the original sequence files to the k-mer set indices (computation of Bloom filters, CQF (Squeakr), Othello). The maximum external memory column corresponds to the peak disk usage when building the index. The time column is the time required to build the set of k-mer sets index (on one processor). The index size column is the final index size. BIGSI is not a compressed index, but the authors had explored the possibility to compress using snappy (https://google.github.io/snappy/). Parameters used for the different methods were $\theta = 0.9$ and BF size of 2.10^9 for the SBT methods, $k = 20$ as the k-mer size for all methods, and 34 "log slots" for Mantis from the estimation of their paper.

Supplementary Box 6. Details on the hash-based strategies in color aggregative methods

Here we give details on the set of k-mer sets implementations in the different hash-based strategies.



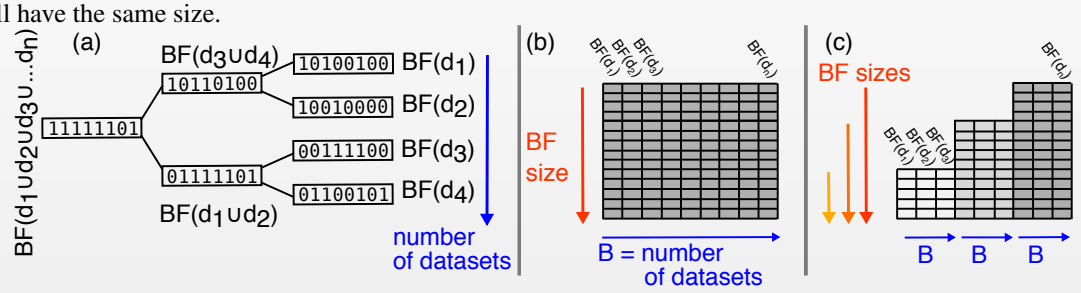
In the Figure above, the terms **compacted DBG** and **unitigs** are defined in Supplementary Box 2. **Minimal Perfect Hash functions (MPHFs)** are functions that map a fixed set of keys to the range of consecutive integers from 0 to the number of keys. They allow to implement memory efficient hash tables (55). For example in Pufferfish, MPHFs are used to associate a k-mer to its position in the graph unitigs. In BLight (42), they associate a k-mer to its position in a bucket.

Some of the techniques use **minimizers**. While there exist multiple definitions in the literature, here we will say that a minimizer is the smallest l-mer that appears within a k-mer, with $l < k$. "Smallest" should be understood in terms of lexicographical order. For example in the k-mer GAACT, the minimizer of size 3 is AAC, as all other l-mers (GAA, ACT) are higher in the lexicographical order than AAC. Minimizers are used here to create partitions of k-mers, which roughly have the same size. Such partitioning techniques reduce the footprint of position encoding.

Supplementary Box 7. k-mer aggregative methods

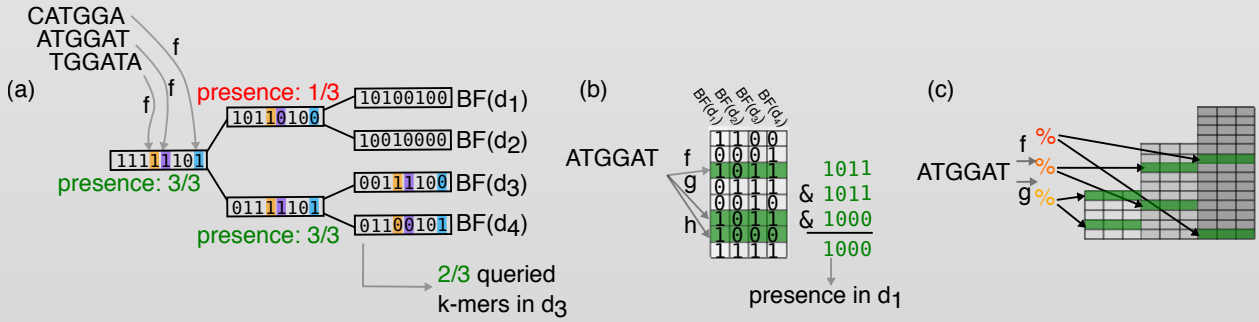
Index construction In SBT ((a) in Figure below), all BFs in leaves are initialized to the same uncompressed size, which means that the structure is initialized according to the estimated total number of distinct k-mers to index (which is, in practice, estimated by k-mer counting techniques applied on a random sample of datasets). As it can be seen in the example Figure, union BFs in parent nodes can thus be simply deduced by applying a logical OR on the children BF pair.

In BIGSI ((b) in Figure below), all initial BFs must have the same size, so this parameter is usually adjusted according to the sample with the highest k-mer cardinality. It is also required that the same hash functions would be used during the construction (in practice 2-5 functions, which allows rapid query). COBS (c) uses the same principle but Bloom filters do not all have the same size.



Queries in tree structures (SBT, etc.) The query of a set of k-mers Q starts at the root node and is propagated through the tree ((a) in Figure below shows the query for 3 k-mers). At each node, the corresponding BF locations are checked to contain 1s or not. If a relative proportion of at least θ of k-mers are returned as present by the BF of the current node (in the example, $\theta = 2/3$), the query proceeds to the children nodes. Otherwise, the children of the current node are not explored. In the Figure below, the query continues after the root node, stops at the subtree of the node which k-mer presence is only at 1/3, and continues in the subtree of the other node. Finally, the procedure stops when tree leaves are reached. The corresponding samples are then returned as containing similar k-mer content with the query (red dataset in the Figure).

Queries in matrix structures (BIGSI) We present the query step for one k-mer. A given k-mer is hashed, leading to one or several rows to lookup. In Figure (b) below, the query is performed on the green rows. Each queried row informs on the datasets that may contain the query k-mer. All the returned bit vectors are then summarized vertically into a single vector, using a logical AND operation. Positions yielding 1s after this operation correspond to datasets that contain the k-mer (in the Figure example, the k-mer is present only in dataset 1). (c) The same principle is used when matrices of several sizes contain the BFs. Hashes are simply adapted to the different sizes using a modulo.

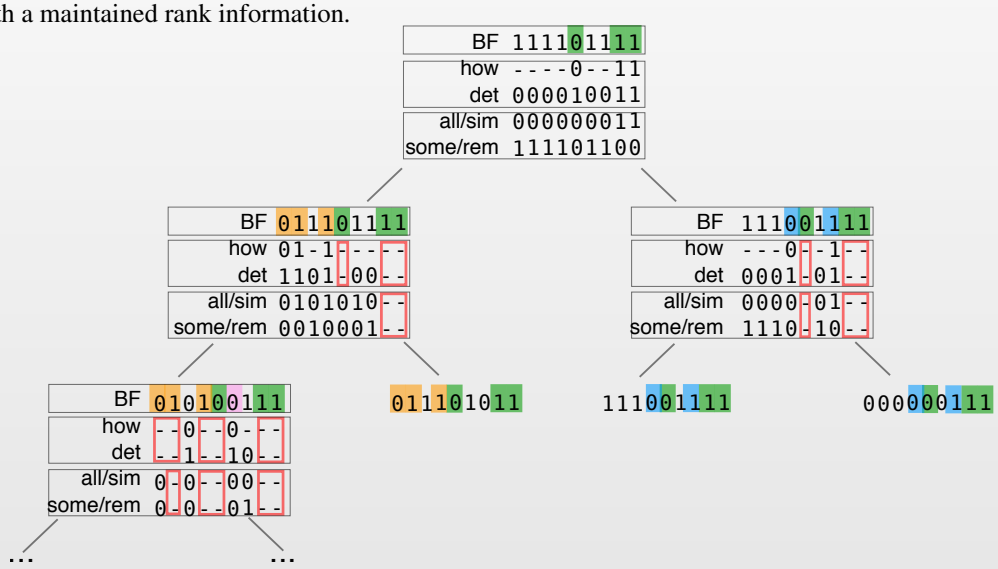


Supplementary Box 8. Contrasts between the different k-mer aggregative approaches

SBT approaches For SBTs, different strategies are used to store information in each node. In the example below, the first level of each node is a plain Bloom filter, as used in the original SBT approach. The second level is the $how + det$ representation used in HowDeSBT. The third is the equivalent $all + some$ or $sim + rem$ representations used by AllSomeSBT and SSBT. The three approaches are shown in four nodes.

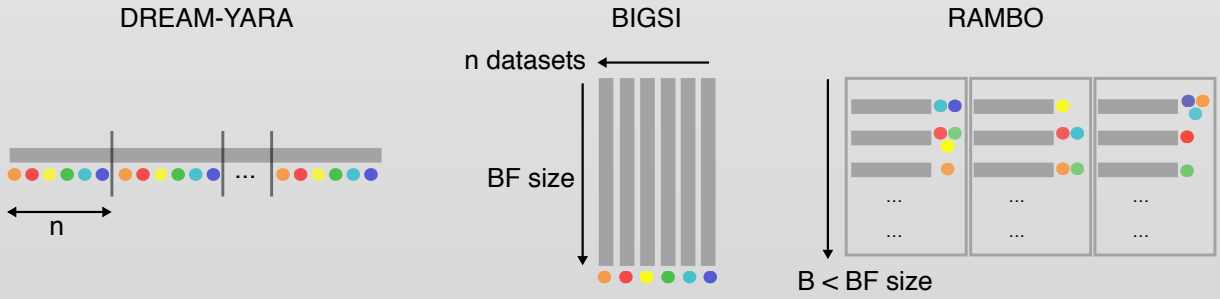
Starting at the root node, some bits are already fixed as they keep the same value in all the nodes of the subtree (green bits). Those bits are marked as det, and when they are, how records their values. In the $sim + rem$ or $all + some$ system, fixed bits with value set to 1 are marked by all and sim. The $some$ or rem vector stores values such that $all \cup some = BF$ or $sim \cup rem = BF$.

At the second level, new bits are fixed (orange in the left subtree and blue in the right subtree). The same rules apply. Moreover, bits that were marked in the upper levels are non informative (red stroke). They can be removed from the structure with a maintained rank information.



Bloom filter arrays Left: the DREAM-Yara index is built by interleaving the bits of each Bloom filters. Bits of the same rank are grouped together in bins of size n. These groups are concatenated to obtain a large vector of size $n \times (t+1)$. Middle: BIGSI matrix built for each dataset (color dots), gray boxes represent BFs. Right: RAMBO is represented as columns of merged BFs.

A query corresponds to a set of rows (in BIGSI) or a set of bins (in DREAM-Yara), which are then combined to obtain the result.



Tool	Max Ext. Memory (GB)	Time (h, wall clock)	Peak RAM (GB)	Index Size (GB)
SBT	N/A	7.7^a	44^a	79^a
SSBT	N/A	32^a	6.1^a	13^a
AllsomeSBT	N/A	4.0^a	29^a	85^a
HowDeSBT	N/A	82^a	430^a	7.6^a
BIGSI	110^a	5.0^a	$1{,}000^a$	-
COBS	12^a	0.05^a	11^a	-
Vari	$1{,}000^b$	11^b	136^b	51^b
Vari-Merge	$1{,}000^b$	12^b	52^b	51^b
Rainbowfish	$1{,}000^b$	11^b	136^b	51^b
BFT	900^b	52^b	120^b	99^b
Multi-BRWT	$1{,}300^b$	42^b	156^b	$1{,}300^b$
Mantis+MST	36^b	12^b	52^b	51^b

Table S5. Space and time required to build indices on bacterial datasets (extrapolated to 4,000 datasets). The table combines results from a benchmark done in the COBS article (45) (denoted by ^{a} in the table) and one from the Vari-Merge article (16) (denoted by ^{b} in the table). The COBS benchmark contains 1,000 microbial DNA files , consisting of various bacterial, viral and parasitic WGS datasets (in the ENA as of December 2016) with an average of 3.4 million distinct k-mers per file. No cutoff on k-mer abundances was used before constructing the data structures. The Vari-Merge benchmark contains 4,000 datasets totalling 1.1 billion distinct k-mers from 16,000 Salmonella strains (NCBI BioProject PRJNA18384). Note that the Vari-Merge benchmark has more genomes than the COBS benchmark, but it possibly contains a lower variability in k-mer content. In the Vari-Merge benchmark, methods were run with $k = 32$, with the exception of BFT that was run with $k = 27$. In the COBS benchmark, k was set to 31. When applicable, other parameters were set by default. In the table, COBS denotes for the COBS *compact* index that allows more than one batches of BF. BIGSI stands for the original implementation of BIGSI. A dash stands for a non-applicable measure (for instance, BIGSI has no compressed size since its not a compressed index). Note also that COBS index is not needed to be fully loaded in RAM even if all results present fully loaded indices.